T-Swap Audit Report

Version 1.0

Cyfrin.io

January 15, 2026

T-Swap Audit Report

fervidflame

January 15, 2026

Prepared by: fervidflame Lead Auditors: fervidflame

Table of Contents

- Medium
  - **[M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline**
- Low
  - **[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order.**
  - **[L-2] Default value return by `TSwapPool::swapExactInput` results in incorrect return value given.**
- Informational
  - **[I-1] Custom error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used, therefore, it should be removed.**
  - **[I-2] `PoolFactory::constructor` laking zero address check.**
  - **[I-3] `PoolFactory::createPool` should use .symbol() instead of .name()**
  - **[I-4] `T-Swap` events should be indexed.**
  - **[I-5] `TSwapPool::constructor` laking zero address check - `wethToken` & `poolToken`.**
  - **[I-6] In `TSwapPool::deposit`, the local variabe, `liquidityTokensToMint`, should be above the external call, `_addLiquidityMintAndTransfer`, to follow CEI.**
  - **[I-7] Avoid using magic numbers and large numeric values in code.**
- Gas
  - **[G-1] Emitting a constant variable in the custom error, `TSwapPool::TSwapPool__WethDepositAmountTooLow` is unnessary and a waste of gas.**
  - **[G-2] Local variable, `poolTokenReserves`, in `TSwapPool::deposit` is unused and should be removed to not waste gas.**

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset. It is similar to Uniswap.

Disclaimer

The fervidflame team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Table 1: We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
Commmit Hash: 1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

- In Scope:

```
./src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum

Roles

- Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.
- Users: Users who want to swap tokens.

Executive Summary

Issues found

Table 2: List and quantity of severities.

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	7
Gas	2
Total	17

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: The getInputAmountBasedOnOutput function is intended to calculate the amount of tokens a user should deposit given an amount of tokens of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: The protocol's econonmic model (the fee structure) is broken and it takes more fees than expected from users.

Proof of Concept (Proof of Code):

Proof of Code

To test this, include the following code in the TSwapPool.t.sol file:

```solidity
function testFlawedSwapExactOutput() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
    poolToken.approve(address(pool), initialLiquidity);

    pool.deposit({
        wethToDeposit: initialLiquidity,
        minimumLiquidityTokensToMint: 0,
        maximumPoolTokensToDeposit: initalLiquidity,
        deadline: uint64(block.timestamp)
    });
    vm.stopPrank();


    // User has 11 pool tokens
    address someUser = makeAddr("someUser");
    uint256 userInitialPoolTokenBalance - 11e18;
    poolToken.mint(someUser, userInitialPoolTokenBalance);
    vm.startPrank(someUser);

    // Users buys 1 WETH from the pool, paying with pool tokens
    // In the protocol, `getInputAmountBasedOnOutput` is not directly
       called. it is called by the function `swapExactOutput`.
    poolToken.approve(address(pool), type(uint256).max);
    pool.swapExactOutput(
        poolToken,
        weth,
        1 ether,
        uint64(block.timestamp)
    );

    // Initial liquidity was 1:1, so user should have paid ~1 pool
       token
    // However, it spent much more than that. The user started with 11
       tokens, and now only has less than
    assertLt(poolToken.balanceOf(someUser), 1 ether);
    vm.stopPrank();

    // The liquidity provider can rug all funds from the pool now,
       including those deposited by user.
    vm.startPrank(liquidityProvider);
    pool.withdraw(
        pool.balanceOf(liquidityProvider),
        1, // minWethToWithdraw
        1, // minPoolTokensToWithdraw
```

```
            uint64(block.timestamp)
        );

        assertEq(weth.balanceOf(address(pool)), 0);
        assertEq(pool.Token.balanceOf(address(pool)), 0);
}
```

Recommended Mitigation:

```
function getInputAmountBasedOnOutput(

uint256 outputAmount,

uint256 inputReserves,

uint256 outputReserves

)

public

pure

revertIfZero(outputAmount)

revertIfZero(outputReserves)

returns (uint256 inputAmount)

{

- return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
    outputAmount) * 997);

+       return ((inputReserves * outputAmount) * 1_000) / ((
   outputReserves - outputAmount) * 997);

}
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: The `swapExactOutput` function does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market conditions change before the transaction processes, the user could get a much worse swap.

Proof of Concept (Proof of Code): To test this, include the following code in the TSwapPool.t.sol file:

Proof of Code

1. The price of 1 WETH right now is 1,000 USDC
2. User inputs a `swapExactOutput` looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offer a `maxInputamount`
4. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 1 WETH is now 10,000 USDC. 10x more than the user expected
5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC, resulting in them paying 10x more than the user expected

```


function testSlippageInSwapExactOutput() public {
            // Liquidity provider approves their tokens to the pool contract
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        // Deposit liquidity into the pool via a liquidity provider

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // Initilize a user with 11 pool tokens

        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);

        // Initilize another user with 100_000 pool tokens
        address richUser = makeAddr("richUser");
        uint256 richUserInitialPoolTokenBalance = 100000e18;
        poolToken.mint(richUser, richUserInitialPoolTokenBalance);
        vm.startPrank(richUser);

        // We get the price of WETH in poolTokens

        uint256 originalWethPrice = pool.getPriceOfOneWethInPoolTokens
            ();
        console.log("The original weth price is", originalWethPrice);

        // User 1 wants to buy 1 WETH from the pool, expecting the
            original weth price, but then a rich user purchases a lot of
             WETH
        // This results in an increase of the price of WETH, and User 1
             has to pay the higher price.

        // Rich user swaps and purchases a lot of WETH
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1000 ether, uint64(block.
            timestamp));
        vm.stopPrank();

            // Check new price of WETH
            uint256 newWethPrice = pool.getPriceOfOneWethInPoolTokens()
                ;

            console.log("The new weth price is", newWethPrice);

        // User 1 swaps
        vm.startPrank(someUser);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1 ether, uint64(block.
            timestamp));
        vm.stopPrank();


        uint256 wethBalanceOfSomeUser = weth.balanceOf(someUser);
        uint256 poolTokenBalanceOfSomeUser = poolToken.balanceOf(
            someUser);

        console.log("wethBalanceOfSomeUser; ", wethBalanceOfSomeUser);
        console.log("poolTokenBalanceOfSomeUser: ",
            poolTokenBalanceOfSomeUser);

            assert(newWethPrice > originalWethPrice);

    }
```

Recommended Mitigation: We should include a `maxInputAmount` so the user only has to spend up to a specific amount, and can predict how much they will spend on the protocol.

```
function swapExactOutput(

IERC20 inputToken,

+       uint256 maxInputAmount,

.

.

.

inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves,
    outputReserves);

+       if(inputAmount > maxInputAmount){

+           revert();

+       }

_swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens.

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the swapped amount.

This is due to the fact that the `swapExactOutput` function is called, whereas the `swapExactInput` function is the one that should be called. Because users specify the exact amount of input tokens, not output.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Proof of Concept (Proof of Code): To test this, include the following code in the TSwapPool.t.sol file:

Proof of Code

```

function testIncorrectParameterOnSellPoolToken() public {
        // Liquidity provider approves their tokens to the pool
            contract
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        // Deposit liquidity into the pool via a liquidity provider

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // Initilize a user with 11 pool tokens

        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        weth.mint(someUser, userInitialPoolTokenBalance);

        // User approves their pool tokens to the pool contract and
           calls `sellPoolToken`

        vm.startPrank(someUser);
        poolToken.approve(address(pool), userInitialPoolTokenBalance);
        weth.approve(address(pool), userInitialPoolTokenBalance);
        pool.sellPoolTokens(1e18);
        vm.stopPrank();

        uint256 finalUserPoolTokenBalance = poolToken.balanceOf(
           someUser);
        uint256 finalUserWethBalance = weth.balanceOf(someUser);
        console.log("Final User Pool Token Balance:",
           finalUserPoolTokenBalance);
        console.log("Final User WETH Balance:", finalUserWethBalance);
    }
```

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be passed to `swapExactInput`)

Additionally, it might be wise to add a deadline to the function, as there is currently no deadline. (MEV later)

```
function sellPoolTokens(

uint256 poolTokenAmount,

+        uint256 minWethToReceive,

) external returns (uint256 wethAmount) {

- return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount,
  uint64(block.timestamp));

+         return swapExactInput(i_poolToken, poolTokenAmount,
  i_wethToken, minWethToReceive, uint64(block.timestamp));

}
```

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`.

Description: The protocol follows a strict invariant of `x * y = k`. Where:

- `x`: The balance of the pool token
- `y`: The balance of WETH
- `k`: The constant product of the two balances

This means, that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hence the `k`. However, this is broken due to the extra incentive in the `_swap` function. Meaning that over time the protocol funds will be drained.

The following block of code is responsible for the issue.

```
swap_count++;

if (swap_count >= SWAP_COUNT_MAX) {

swap_count = 0;

outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);

}
```

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collecting the extra incentive given out by the protocol.

Most simply put, the protocol's core invariant is broken.

Proof of Concept (Proof of Code):

Proof of Code

1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens

2. That user continues to swap until all the protocol funds are drained.

Place the following into TSwapPool.t.sol.

```
function testInvariantBroken() public {

vm.startPrank(liquidityProvider);

weth.approve(address(pool), 100e18);

poolToken.approve(address(pool), 100e18);

pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));

vm.stopPrank();

uint256 outputWeth = 1e17;

vm.startPrank(user);

poolToken.approve(address(pool), type(uint256).max);

poolToken.mint(user, 100e18);

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

int256 startingY = int256(weth.balanceOf(address(pool)));

int256 expectedDeltaY = int256(-1) * int256(outputWeth);

pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
    timestamp));

vm.stopPrank();

uint256 endingY = int256(weth.balanceOf(address(pool)));

int256 actualDeltaY = int256(endingY) - int256(startingY);

assertEq(actualDeltaY, expectedDeltaY);

}
```

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
- swap_count++;
- // Fee-on-transfer
- if (swap_count >= SWAP_COUNT_MAX) {
- swap_count = 0;
- outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
- }
```

Medium

[M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline

Description: The `deposit` function accepts a deadline parameter, which according to the documentation is “The deadline for the transaction to be completed by”. However, this parameter is never used. As a consequence, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transactions could be sent when market conditions are unfavorable, even when adding a deadline parameter.

Proof of Concept (Proof of Code): The `deadline` parameter is unused.

From Solidity complier:

```
Warning (5667): Unused function parameter. Remove or comment out the
    variable name to silence this warning.
```

```
   --> src/TSwapPool.sol:103:9:
    |
103 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Recommended Mitigation: Consider making the following change to the function:

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
        {...}
```

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order.

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTrans` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Off-chain functions will receive incorrect event emission information, which could potentially result in malfunctioning.

Recommended Mitigation:

```
    function _addLiquidityMintAndTransfer(...
    ) private {

        ...

-         emit LiquidityAdded(msg.sender, poolTokensToDeposit,
   wethToDeposit);
+         emit LiquidityAdded(msg.sender, wethToDeposit,
   poolTokensToDeposit);

        ...

    }
```

[L-2] Default value return by `TSwapPool::swapExactInput` results in incorrect return value given.

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `output`, it is never assigned a value, nor uses an explicit return statement.

Impact: The return value will always be `0`, giving incorrect information to the caller.

Recommended Mitigation:

```
{

uint256 inputReserves = inputToken.balanceOf(address(this));

uint256 outputReserves = outputToken.balanceOf(address(this));

- uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);

+         output = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);

- if (outputAmount < minOutputAmount) {
- revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);

+         if (output < minOutputAmount) {

+             revert TSwapPool__OutputTooLow(output, minOutputAmount);

}

- _swap(inputToken, inputAmount, outputToken, outputAmount);

+         _swap(inputToken, inputAmount, outputToken, output);

}

}
```

Informational

[I-1] Custom error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used, therefore, it should be removed.

Recommended Mitigation:

```
-     error PoolFactory__PoolDoesNotExist(address tokenAddress);
```

[I-2] `PoolFactory::constructor` laking zero address check.

Recommended Mitigation:

```
    constructor(address wethToken) {
+   if(wethToken == address(0)){
+    revert();
+}
i_wethToken = wethToken;
    }
```

[I-3] `PoolFactory::createPool` should use .symbol() instead of .name()

Recommended Mitigation:

```
- string memory liquidityTokenSymbol = string.concat("ts", IERC20(
   tokenAddress).name());
+ string memory liquidityTokenSymbol = string.concat("ts", IERC20(
   tokenAddress).symbol());
```

[I-4] T−Swap events should be indexed.

Recommended Mitigation:

```
-     event Swap(address indexed swapper, IERC20 tokenIn, uint256
   amountTokenIn, IERC20 tokenOut, uint256 amountTokenOut);
+     event Swap(address indexed swapper, IERC20 indexed tokenIn,
   uint256 amountTokenIn, IERC2 indexed tokenOut, uint256
   amountTokenOut);
```

[I-5] `TSwapPool::constructor` laking zero address check - `wethToken` & `poolToken`.

Recommended Mitigation:

```
constructor(address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    )
        ERC20(liquidityTokenName, liquidityTokenSymbol)
```

```
{
+ if(wethToken || poolToken == address(0)){
+     revert();
+     }
i_wethToken = IERC20(wethToken);
i_poolToken = IERC20(poolToken);
}
```

[I-6] In `TSwapPool::deposit`, the local variabe, `liquidityTokensToMint`, should be above the external call, `_addLiquidityMintAndTransfer`, to follow CEI.

Description:

```
function deposit(...) external revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint){

    ...


    if (totalLiquidityTokenSupply() > 0) {
        ...
}   else {
        _addLiquidityMintAndTransfer(wethToDeposit,
           maximumPoolTokensToDeposit, wethToDeposit);
        liquidityTokensToMint = wethToDeposit;
        }
        }
```

To follow best practices for security, it is recommened to follow CEI (Checks, Effects, and Interactions). Therefore the external call, `_addLiquidityMintAndTransfer`, should below the local variable, `liquidityTokensToMint`.

Recommended Mitigation:

```
function deposit(...) external revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint){

    ...


    if (totalLiquidityTokenSupply() > 0) {
        ...
}   else {
+       liquidityTokensToMint = wethToDeposit;
        _addLiquidityMintAndTransfer(wethToDeposit,
           maximumPoolTokensToDeposit, wethToDeposit);
-        liquidityTokensToMint = wethToDeposit;
        }
        }
```

[I-7] Avoid using magic numbers and large numeric values in code.

Description:

All number literals should be replaced with constants, and all large numeric values should be in scientitic notatio. This makes the code more readable and easier to maintain. Numbers without context are called magic numbers. For example, `uint256 denominator = (inputReserves * 1000)+ inputAmountMinusFee`; in the `TSwapPool::getOutputAmountBasedOnInput` function.

Recommended Mitigation: For magic numbers, assign the number to a constant variable that explains the reason for the number.

For large numeric values, use scientific notation, assign it to a constant variable, and name it like you would for a magic number.

```
+   uint256 public constant ONE_WETH_OR_POOLTOKEN = 1e18;
+   uint256 public constant FEE_PERCENTAGE = 997;
+   uint256 public constant TOTAL_PERCENTAGE = 1000;
+   uint256 private constant MINIMUM_WETH_LIQUIDITY = 1e9;


-    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
-    uint256 inputAmountMinusFee = inputAmount * 997;
-    uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee
    ;
-    getOutputAmountBasedOnInput(
                 1e18, i_wethToken.balanceOf(address(this)),
    i_poolToken.balanceOf(address(this))
-            );
-    getOutputAmountBasedOnInput(
                 1e18, i_poolToken.balanceOf(address(this)),
    i_wethToken.balanceOf(address(this))
-            );
-      outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
-      return ((inputReserves * outputAmount) * 10000) / ((
    outputReserves - outputAmount) * 997);


+    uint256 inputAmountMinusFee = inputAmount * FEE_PERCENTAGE;
+    uint256 denominator = (inputReserves * TOTAL_PERCENTAGE) +
    inputAmountMinusFee;
+    getOutputAmountBasedOnInput(
                 ONE_WETH_OR_POOLTOKEN, i_wethToken.balanceOf(address(
    this)), i_poolToken.balanceOf(address(this))
+            );
+    getOutputAmountBasedOnInput(
+                ONE_WETH_OR_POOLTOKEN, i_poolToken.balanceOf(address(
    this)), i_wethToken.balanceOf(address(this))
+            );
+     outputToken.safeTransfer(msg.sender, ONE_WETH_OR_POOLTOKEN);
+     return ((inputReserves * outputAmount) * TOTAL_PERCENTAGE) / ((
    outputReserves - outputAmount) * FEE_PERCENTAGE);
```

Gas

[G-1] Emitting a constant variable in the custom error, `TSwapPool::TSwapPool__WethDepositAmountTooLow` is unnessary and a waste of gas.

Description:

```
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
```

```
if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
s            revert TSwapPool__WethDepositAmountTooLow(
    MINIMUM_WETH_LIQUIDITY, wethToDeposit);
}
```

`MINIMUM_WETH_LIQUIDITY` is a constant variable and anyone could check this variable (even though it's private) by reading the bytecode. Emitting it uses unnecessary gas.

Recommended Mitigation:

```
if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
-            revert TSwapPool__WethDepositAmountTooLow(
    MINIMUM_WETH_LIQUIDITY, wethToDeposit);
+            revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
}
```

[G-2] Local variable, `poolTokenReserves`, in `TSwapPool::deposit` is unused and should be removed to not waste gas.

Recommended Mitigation: Consider making the following change to the function.

```
function deposit(...) external revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint){

    ...


    if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this));
-           uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));

    ...

}
        }
```